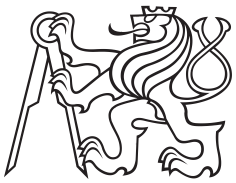
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Scalable Representations of Neural Networks

David Pavlíček

Supervisor: Ing. Zdeněk Buk, Ph.D.
Field of study: Computer and Information Science
May 2017

Acknowledgements

I would like to thank my supervisor Ing. Zdeněk Buk, Ph.D. for his guidance and inspiration. I would also like to thank my family and friends for the support they gave me.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university thesis.

.....
David Pavlíček
Prague, 25 May 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

.....
David Pavlíček
V Praze, 25. května 2017

Abstract

This bachelor's thesis looks at methods for indirect encoding of neural network weights in evolutionary algorithms with focus on network scalability. HyperGP algorithm was implemented to generate neural networks used to control a robot navigating a simulated environment in order to compare multiple approaches to the evolution. The first approach gradually increases the array of sensors and/or neurons in the controlling network during evolution. The second runs the evolution with full sensor and neural network density from the start. Multiple experiments were conducted to explore the impact of these two approaches on the time it takes the algorithm to develop solutions of desired quality.

Keywords: indirect encoding, artifiical neural network, evolutionary algorithm, genetic porogramming

Abstrakt

Tato bakalářská práce se zaměřuje na metody nepřímého kódování neuronových sítí pro evoluční algoritmy s důrazem na škálovatelnost sítě. Je implementován algoritmus HyperGP k evoluci sítí použitých k řízení pohybu robota v simulovaném prostředí za účelem porovnání různých přístupů k evoluci. První přístup postupně navyšuje počet senzorů a neuronů v síti během evoluce. Druhý přístup v evoluci využívá plný počet senzorů a neuronů již od začátku. Bylo provedeno několik experimentů s cílem porovnat tyto přístupy.

Klíčová slova: nepřímé kódování, umělá neuronová síť, evoluční algoritmy, genetické programování

Překlad názvu práce: Škálovatelná reprezentace neuronových sítí

Contents

Chapter 1

Introduction

Artificial Neural Networks (ANNs) are a powerful tool that have received a major wave of interest in recent years. There is a lot of progress being made, most notably in the field of "deep learning". Neural Networks are being used to solve many difficult problems such as Computer Vision, Natural Language Processing, Machine Language Translation, Machine Learning, Data Mining, Stock Market Prediction, Physical System Modeling, Knowledge Discovery, Games Playing. The training of these networks has recently been enabled by multiple factors most notably by the availability of cheap computational resources typically in the form of Graphical Processing Units (GPUs), enormous amounts of data and multiple algorithmic advances.

The training of ANNs still remains computationally expensive and time consuming. In the case of using evolutionary algorithms to evolve ANNs there are two approaches to encoding the network as a set of genes, either direct or indirect encoding. When using the direct encoding the genome contains information about every network weight and the network structure. This causes the size of the network representation to quickly grow with the size of the network. This results in high dimensionality of the search space and so the evolutionary algorithm convergence is slow. On the other hand, indirect encoding avoids this problem by representing the network in a way that is independent of it's actual size and structure. Indirect encoding holds information about the network in an abstracted form from which we can construct the network. This is done so that the indirect representation is small compared to the network. As a result the search space is greatly reduced which in turn enables the algorithm to converge faster. Several of these methods have been developed some of which are described in Chapter 2.

The independence between indirect encoding and the size of the ANN it represents offers the possibility to generate the network at different scales while preserving its ability to perform the task it has been trained to do [1, 2]. The performance of the network might be somewhat decreased by the scaling process but it has been shown that it can be increased again by further evolution [1, 2]. Main goal of this bachelor's thesis is to compare the effects of scaling up networks evolved at a small scale, increasing the network resolution during evolution and running evolution with high resolution from the start. After discussion with my supervisor I have chosen to implement

HyperGP algorithm [3] to carry out these experiments. The details of the implementation are discussed in Chapter 3.2. The experiments performed are described in Chapter 4.

Chapter 2

Theoretical Background

2.1 Artificial Neural Networks

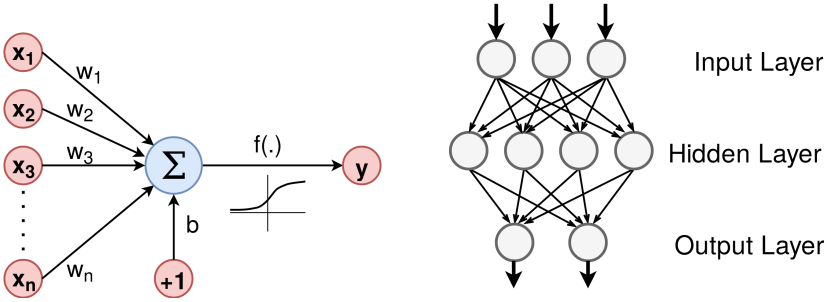
The concept of Artificial Neural Networks takes inspiration from the animal brains in real life. Although ANN is only a crude approximation of the real brain it is an extremely powerful tool applicable to many areas. Several types of ANNs have been developed to day (e.g. feed-forward neural networks, recurrent neural networks, convolutional neural networks, deep neural networks, LSTMs) but the basic idea is well summed up by Simon Haykin in his book [4]:

"A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
1. Knowledge is acquired by the network from its environment through a learning process.
2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge."

The units constituting the network are called neurons. The exact structure of these neurons depends on the type of the network. For example one of the simpler models of the neuron can be represented by an equation (2.1). A single neuron takes a weighted sum of the incoming signals and scales it by an activation function. This is typically some non-linear logistic function e.g. sigmoid function or hyperbolic tangent. Usually, each neuron also has a bias. This can be thought of as abias weight applied to an additional input with value always equal to 1. The resulting value is the neuron's output signal which is then further propagated to the other neurons in the network. The diagram of a simple network and a model of a neuron are shown in Figure 2.1.

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

Where y is the neuron's output, w_i are the input weights, x_i are the input signals, b is the bias and f is the activation function.



(a) : Model of a single neuron.

(b) : Single-layer feed-forward neural network with three input neurons, four hidden neurons and two output neurons.

Figure 2.1: Diagram of a neuron unit and a simple neural network.

2.2 Methods for Indirect Encoding of Neural Networks

Various method for indirectly encoding neural networks have been developed. In this section I briefly describe HyperNEAT, HyperGP and CNCS. There are many others not discussed here .

2.2.1 HyperNEAT

HyperNEAT is a method for the evolution of artificial neural networks which uses indirect encoding introduced by Kenneth Stanley and David D'Ambrosio in [1, 2]. In the papers the authors recognize that "a significant problem for evolving artificial neural networks is that the physical arrangement of sensors and effectors is invisible to the evolutionary algorithm", "sensors and effectors with consistent geometric relationships can be exploited by a repeating motif in the neural architecture," and "exploiting sensor geometry requires a generative encoding because it is necessary to correlate repeated connectivity motifs to regularities in the physical placement of sensors and effectors". They have developed an encoding called connective Compositional Pattern Producing Network (connective CPPN). This encoding can be used to develop complex neural networks containing regularities such as symmetries and repetitions. The neurons have defined positions in a 2D grid called substrate. A connective CPPN represents a four-dimensional function mapping the coordinates of a source and the target neurons to the weight of the connection between them.

The connection is not expressed if the weight is below a certain threshold. An advantage of this approach is that the number of neurons in the grid can be scaled arbitrarily while preserving the general connectivity concept. [2] presents an experiment where an agent equipped with sensors and effectors concentrically positioned around its body was trained to collect food in its environment. The experiment showed that as the number of neurons in the substrate has been increased the generated network still retained its functionality. Although the scaling has caused a small degradation to network's fitness the network has regained the previous fitness after being allowed to continue the evolution for no more than five generations. The CPPN is similar to a neural network, however the nodes in CPPN use various transfer functions (e.g. sigmoid function, gaussian function, sine, cosine, absolute value) which enable it to generate regular patterns. NEAT (Neuroevolution of Augmenting Topologies) algorithm [5] was developed to evolve neural network weights as well as its topology. In HyperNEAT it is instead used to CPPNs which in turn generate neural networks. The NEAT algorithm starts off with simple networks and through process of complexification adds additional neurons and connections. The important feature of NEAT is the niching algorithm. It keeps track of multiple species within the population which allows it to protect new topologies that emerge.

2.2.2 HyperGP

HyperGP algorithm [3] uses the idea of hypercube-based encoding introduced in HyperNEAT [1] but replaces CPPN with CPPF (Compositional Pattern Producing Function) and NEAT with Genetic Programming. The paper [3] presents an experiment where HyperGP and HyperNEAT were compared on the task of controlling a simulated robot. Both algorithms have generated solutions of comparable quality but HyperGP has shown faster convergence than HyperNEAT. The sensors and neurons are placed in a 2D substrate. The CPPF is a mathematical expression using up to four variables as inputs corresponding to the coordinates of the source and target neurons. It is constructed from a set of functions (for example: addition, multiplication, sine, cosine, absolute value, gaussian function), variables and constants. In the experiment from [3] a network is encoded as three separate CPPFs. First encodes the weights between input sensors and neurons, second the weights between neurons and the third biases of the neurons. This is the same setup I have used in my experiments.

2.2.3 CNCS

Compressed Network Complexity Search (CNCS) algorithm has been presented in [6]. In their previous work [7] the authors of the paper have published method for encoding the neural network as Fourier-type coefficients. Evolutionary search thus proceeds in frequency-domain and the weight matrices are then obtained by performing inverse Fourier transform. "If adjacent weights in the matrices are correlated, then this regularity can be encoded using fewer

coefficients than weights, effectively reducing the search space dimensionality. For problems exhibiting a high-degree of redundancy, this “compressed” approach can result in an order of magnitude fewer free parameters and significant speedup"[6]. Previously, when this approach has been applied the network topology and number of coefficients have been fixed. CNCS runs multiple evolutions using different complexity classes. It keeps a probability distribution for these classes which starts biased towards lower complexity. The distribution determines how the run-time is allocated for each of the complexity classes. During the run the algorithm computes the expected fitness of the complexity classes by sampling the population and then adjusts the probability distribution accordingly.

CNCS starts off favoring the simple solutions and gradually moves to more complex ones. This property is similar to that of the HyperNEAT algorithm.

Chapter 3

Implementation

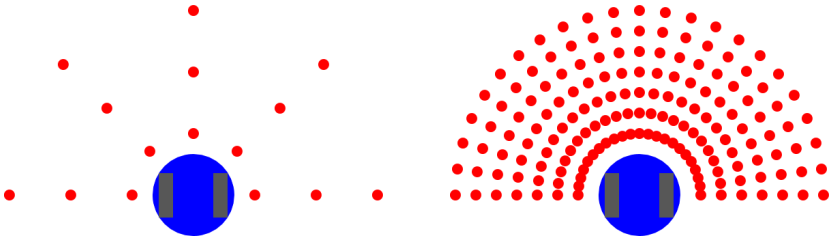
3.1 Simulator

For the purposes of experimentation I have created a simulation environment similar to that used in [3] and [8]. The simulator is nescessary to obtain the fitness of the individuals in the evolution. Both the simulator and the HyperGP algorithm have been implemented in Java. The source code, compiled JAR file as well as configuration used in the experiments can be found on the CD accompanying this document.

3.1.1 Simulation Setup

Each of the evolved neural networks is evaluated on its ability to control a two-wheeled robot navigating across a map. The robot drives around in the simulation for a set period of time and its average speed is recorded. To obtain fitness of an individual its average speed is then divided by the maximum speed the robot can achieve. There are two types of surfaces: road and grass. The robot moves five times faster on a road then he does on the grass. This causes the algorithm to evolve networks capable of keeping the robot on the road while also increasing its speed.

In order to behave intelligently, the robot needs to obtain information from its environment. There are various types of sensors such as distance sensors, contact sensors, accelerometers, gyroscopes, GPS, and many others. Since there are neither obstacles nor other agents in this simulation I have decided to use sensors which can determine the type of surface around the robot. They are arranged in an 180 degree wide array in front of the robot in concentric semicircles (see Figure 3.1). The size of this array can be scaled up to arbitrary resolution. The sensors detect the color of the surface and map the grass (green) to 0 and the road (gray) to 1. This information is then used as input for the neural network.



(a) : Sprase sensor array **(b) :** Dense sensor array

Figure 3.1: Diagram of a robot equipped with two wheels and an array of sensors at two different resolutions.

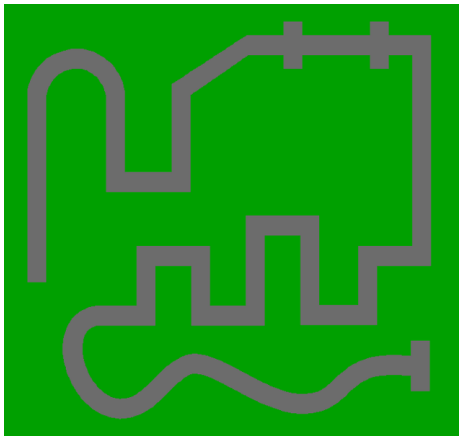


Figure 3.2: Example of one of the maps used in the experiments.

3.1.2 Neural Network Setup

Once the robot has the means of gathering information from its environment it also needs a brain in order to react to the received information and to navigate successfully. In this case the brain consists of a single layer of neurons forming recurrent neural network. Each neuron is a mathematical function modeled on real-life neurons. The network I have implemented is a fully recurrent neural network described in [9]. The full dynamics of the whole network with n units and m external inputs is described by the three equations below where $\mathbf{y}(t)$ is a n-tuple of outputs of the neurons and $\mathbf{x}(t)$ is a m-tuple of input signals at time t (this includes additional input which is always 1 for bias). $\mathbf{x}(t)$ and $\mathbf{y}(t)$ are concatenated into a $(n+m)$-tuple $\mathbf{z}(t)$:

$$z_k(t) = \begin{cases} x_k(t), & \text{if } k \in I \\ y_k(t), & \text{if } k \in U. \end{cases} \tag{3.1}$$

I is a set of indices for which z_k is an external input and U is a set of indices for which z_k is an output of a neuron. $\mathbf{W}$ is a weight matrix for the network.

$$s_k = \sum_{l \in U \cup I} w_{kl} z_l(t) \tag{3.2}$$

s_k denotes the net input to the kth neuron at time t, for $k \in U$. The unit's output at the next time step is:

$$y_k(t+1) = f_l(s_k(t)) \tag{3.3}$$

It is possible for every neuron to be connected to all others including recurrent connections but this type of connection does not always have to be the case. The process of generating the connections and their weights is described in detail in 3.2.4. For the neural network no parallel computation was used. The code for the neural network can be seen in C.1.

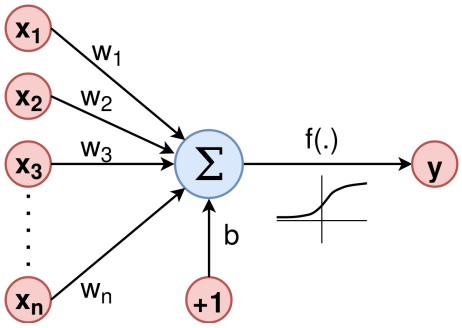


Figure 3.3: Single neuron unit.

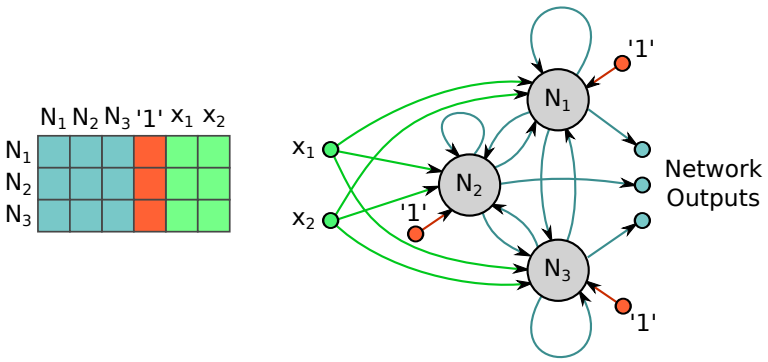


Figure 3.4: Diagram [10] of a small fully recurrent neural network with two inputs and three neurons.

3.2 HyperGP

3.2.1 Genetic Programming

In their book[11] Riccardo Poli, William B. Langdon and Nicholas F. McPhee offer description of Genetic Programming:

"Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. At the most abstract level GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level statement of what needs to be done."

"In genetic programming we evolve a population of computer programs. That is, generation by generation, GP stochastically transforms populations of programs into new, hopefully better, populations of programs, cf. Figure 1.1. GP, like nature, is a random process, and it can never guarantee results. GP's essential randomness, however, can lead it to escape traps which deterministic methods may be captured by. Like nature, GP has been very successful at evolving novel and unexpected ways of solving problems."

Algorithm 1: Basic genetic algorithm

```
Input: maximum number of runs M, population size N
Output: best individual
for run ← 1 to M do
    gen ← 0;
    repeat
        if gen = 0 then
            populaion ← ∅;
            offspring ← generateRandom(N);
        else
            offspring ← generateOffspring(N, populaion);
        end
        evaluate(offspring);
        populaion ← select N best from populaion ∪ offspring;
        gen ← gen + 1;
    until termination condition;
    return best individual;
end
```

The genetic programming algorithm requires several parameters to control its behavior. The first one is the *population size* which determines the num-

ber of individuals in the population. The probabilities of genetic operations such as $p_{mutation}$, $p_{crossover}$ influence the way offspring are generated. The *maximum tree depth* limits the size of individual representation. There can be parameters determining the selection pressure in selection function e.g. size of the selection pool in tournament selection. We can run the evolution multiple times if we set *number of runs* greater than 1. Evolutionary algorithms use a **selection method** to determine which individuals from the parent generation enter the mating pool. Genetic operators are then applied to the individuals in the mating pool to create the next generation. It is essential that the selection method favors the individuals with higher fitness. The selection pressure is a property of the selection method describing how much more the fitter individuals are favored. If the pressure is too high the algorithm converges prematurely and gets "stuck" in a local optimum. On the other hand if the pressure is too low the algorithm might take too long find a suitable solution. Two of the commonly used ones are **roulette wheel selection** and **tournament selection**.

Roulette wheel selection, also known as fitness proportionate selection, stochastically selects an individual from a population of size N with probability:

$$p_k = \frac{fitness(k)}{\sum_{i=1}^{N} fitness(i)} \tag{3.4}$$

Tournament selection is a method which first randomly selects a subset P of k individuals from the population. These individuals are then compared with each other and the one with the highest fitness is selected. The selection pressure can be adjusted by changing the size of the selection pool k [12].

$$\underset{x \in P}{\operatorname{argmax}}(fitness(x)) \tag{3.5}$$

We also need to provide the algorithm with some termination conditions to know when to stop the run. This can bethis can be for example reaching the *maximum number of generations* or obtaining a solution of desired quality. The algorithm then returns the best individual that has been so far developed.

3.2.2 Genome Representation

Each individual genome in the population is composed of three different CPPFs. First (f_i) is for encoding input weights, second (f_n) for connection weights between every two neurons and third (f_b) encodes neuron biases. CPPF is a mathematical expression which is a function of up to four arguments and returns a single value [3]. When we are using genetic programming to evolve a function it is useful to represent it as a tree structure. [11]Internal

nodes are assigned primitive functions and leaves terminals (input variables or constants). As the tree grows more complex functions are generated. For example CPPF in Figure 3.5 corresponds to function $f(x_1, y_1, x_2, y_2) = 3.2 \cdot x_2 \cdot y_2 + sin(-x_1)$.

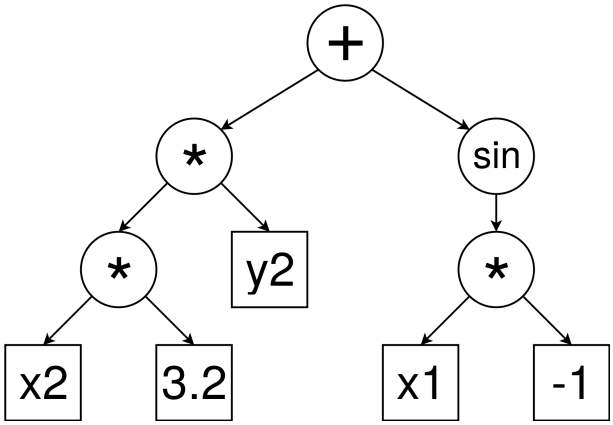


Figure 3.5: CPPF as a tree structure. Circles represent primitive functions while squares represent terminals.

In my implementation I have decided to use the same atoms and primitive functions that were used in [3] and only added squaring function $f(x) = x^2$ and negation $f(x) = -x$. All are listed in Table 3.1.

primitive set	
primitive functions	terminals
$x + y$	x_1
$x \cdot y$	y_1
$sin\ x$	x_2
$cos\ x$	y_2
$tan^{-1}x$	-1
$\sqrt{\lvert x \rvert}$	rand(-5, 5)
$\lvert x \rvert$	
e^{-x^2}	
$e^{-(x-y)^2}$	
x^2	
$-x$	

Table 3.1: Primitive functions and atoms used in CPPFs.

Each CPPF has a defined number of input variables it can use and maximum depth of the tree. CPPFs encoding input-to-neuron and neuron-to-neuron weights work with 4 variables (x_1, y_1, x_2, y_2) while CPPF encoding biases only uses two (x_1, y_1)

In evolutionary algorithms the next generation is created by repeatedly selecting individuals from the current generation by a selection function based

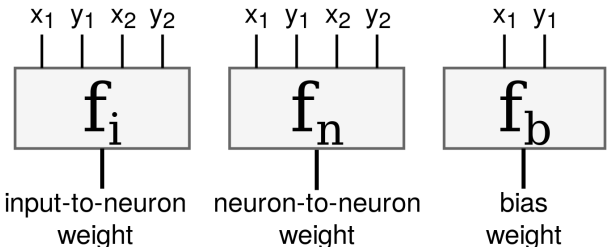
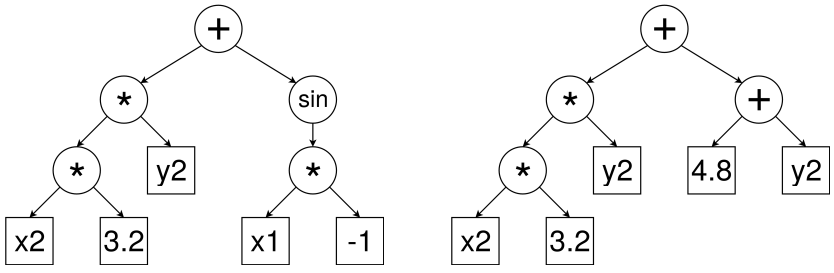


Figure 3.6: Three CPPFs constituting the genome of an individual.

on their fitness. One of genetic operators is then applied to the individual to create a new one that is used in the following generation. There are several of these operators such as *reproduction*, *mutation*, *crossover* and *permutation* [13].Based on the finding in [3] that use of mutation without crossover gives better performance I have decided to use only *reproduction* and *mutation* in my algorithm.

My implementation of the genome representation supports these operations:

- **Random individual creation** constructs three entirely new CPPFs respecting the specified maximum depth for the expression and each with their number of variables utilizing the *grow method* [11]. To construct a new CPPF tree the "nodes are selected from the whole primitive set (i.e., functions and terminals) until the depth limit is reached. Once the depth limit is reached only terminals may be chosen (just as in the full method)."[11]
- **Reproduction** returns an identical copy of the individual.
- **Mutation** selects a random node in one of the three CPPFs and replaces subtree rooted in that node with new randomly generated subtree while respecting defined maximum depth. This operation is sometimes called a *headless chicken mutation*.

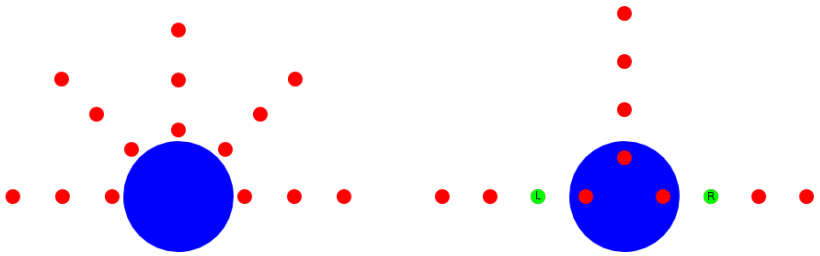


(a) : Before **(b) :** After

Figure 3.7: Example of CPPF mutation operation. Expression in **b)** was created from **a)** by removing subtree rooted in the node labeled *sin* and replaced by a new subtree.

3.2.3 Substrate

The sensors and neurons controling the robot have their definite positions in 2D space. These positions are represented as polar coordinates in a grid that is called *substrate* [1]of which there are two types *sensor substrate* for sensors and *neural substrate* for neurons. The neural substrate contains additional information describing which neurons are used as outputs driving the effectors (wheels).



(a) : Sensor substrate **(b) :** Neural substrate

Figure 3.8: Examples of sensor and neuron substrates. Blue circle is the robot body, red circles are the sensors in **a)**, neurons in **b)** and green circles are the output neurons labeled L (left wheel) and R (right wheel)

3.2.4 Neural Network Generation

To obtain the input-to-neuron, neuron-to-neuron and bias weights, the polar coordinates from substrates are used as inputs for the corresponding CPPFs. For example to get the connection weight between particular sensor and a neuron, we take that sensor's coordinates from the *sensor substrate* and use them as r_1 and φ_1 while using the neuron's coordinates from *neural substrate* as r_2 and φ_2 in the CPPF $f_i(r_1, \varphi_1, r_2, \varphi_2)$ which returns the weight value. For bias generation CPPF f_b requires only the coordinates of a single neuron because the bias concerns only that one neuron. A weight that has an absolute value below a given threshold is not expressed and the other weights are scaled to be between 0 and maximum weight value.

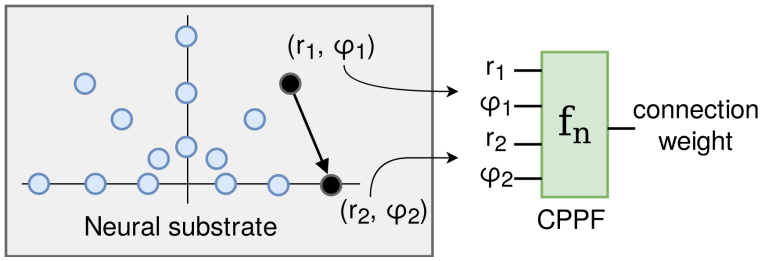


Figure 3.9: The connection weight between every two neurons is determined by passing their coordinates as arguments into CPPF f_n.

3.2.5 Program Controls

The program has an easy to use graphical user interface (GUI) shown in Figure 3.10. The user first specifies all the evolution parameters and substrates in the configuration panels on the left, number of runs to execute and type of sata to export in the export panel on the top right. The evolution is started by the "play" button and the progress as well as several current values are displayed in the middle panel. Once the program has finished, all the results can be found in the project directory. The detailed manual to the program can be found in Appendix B

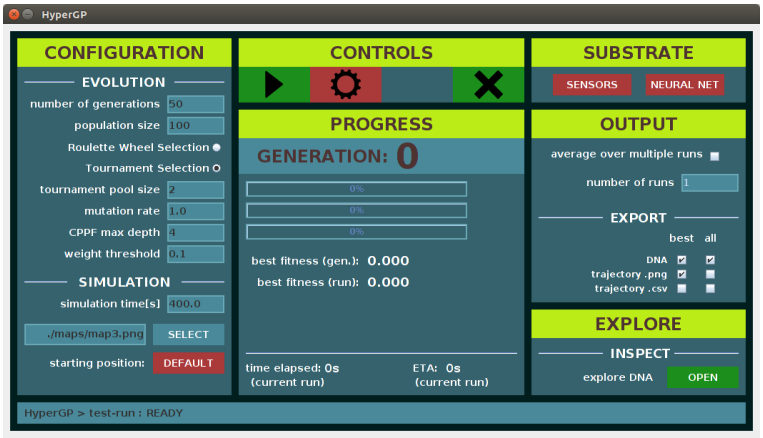


Figure 3.10:

3.2.6 Output

After the algorithm has run a desired number of times the program returns the data from all the runs and saves it in CSV format. This includes average and best fitness for each generation in each run. For convenience it adds an additional column containing this data averaged over all runs. During the run the program saves all the data the user has specified in the project directory. Everything is organized in a directory structure shown in Figure 3.11. It saves the best individuals from each run as well as information about each generation's average as well as best fitness and the fitness of every individual.

The individual is exported into a file with **.dna** extension. The file contains the three CPPFs encoded as text in human-readable form (Figure 3.12).

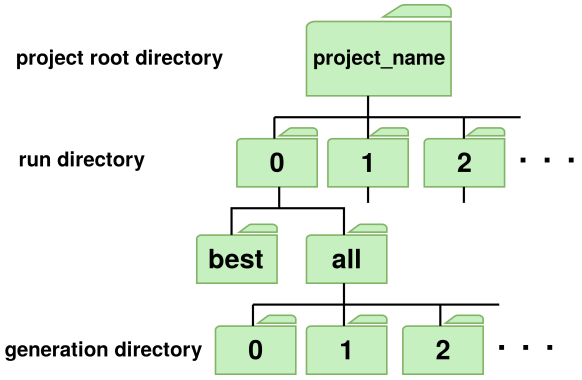


Figure 3.11: Diagram of the project directory structure. The root directory contains subdirectories numbered after the number of a run. In each of these there are two directories: **best** which contains the best results from each generation and **all** containing more subdirectories numbered after generations. Every individual is exported into the directory corresponding to its generation.

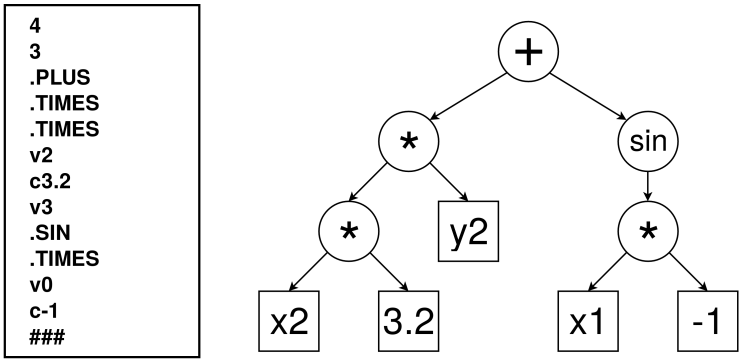


Figure 3.12: CPPF(right) encoded as a text(left). First number defines the number of inputs, second the tree depth. Next lines correspond to nodes in the tree as we traverse it in depth-first fashion. Lines beginning with a dot represent primitive functions, lines beginning with v input variables and lines starting with c constants. A file containing all three CPPFs of the individual will have them appended in this format in order f_i, f_n and f_b all separated by a line starting with #.

Chapter 4

Experimental Results

Main goal of this bachelor's thesis is to compare several approaches to evolution of neural networks using HyperGP algorithm. In this chapter I present the results from the experiments I have conducted. First I have recreated the experiment presented in [3] (section 4.1). I have taken the best evolved solution, scaled the substrates to higher resolutions and evaluated the change in performance. I have then proceeded to compare the evolution using different sensor substrate resolutions (4.3). In section 4.4 I present the results obtained by running the evolution with increasing substrate resolutions. In all the experiments the settings regarding the simulation of the robot have been left the same. Because evolutionary algorithms are inherently stochastic, the program has been run ten times for each of the settings (except for the first experiment). Files from all the experiments can be found on the CD accompanying this document. During the experiments I have observed that when the individual reaches fitness of 0.7 or higher it is able to stay on the road successfully. Therefore I consider the problem to be solved at the fitness of 0.7. In some cases the termination condition has been added where the evolution is stopped when the fitness of 0.9 is reached. The parameters shown in Table 4.1 are the same for all the experiments.

Parameter	Value
population size	100
max. number of generations	50
tournament pool size	2
CPPF tree depth	4
weight threshold	0.1

Table 4.1: Parameters same across all the experiments.

4.1 Recreation of the Original Experiment

4.1.1 Setup

The HyperGP experiment presented in [3] used a small number of sensors and neurons. The sensors were placed in a 3x5 (3 rays of 5 units) substrate and the neurons in a 3x3 (3 rays of 3 units) substrate. (see Figure A.1 **a)** and **b)**). I have used this same setup in my first experiment with the aim to recreate the original experiment. The values of the other parameters are the same as in Table 4.1. I have run the program twenty times for this configuration.

4.1.2 Results

The results are shown in Figure 4.1. From the twenty runs of the algorithm that were executed fifteen achieved the target fitness of 0.7 or higher within fifty generations. The mean fitness reached after fifty generations was 0.735 and the fitness obtained by the best individual was 0.924. Median number of iterations before target fitness was reached is 13.5. My results show slightly worse performance than the original experiment in [3] has produced when using crossover with mutation.

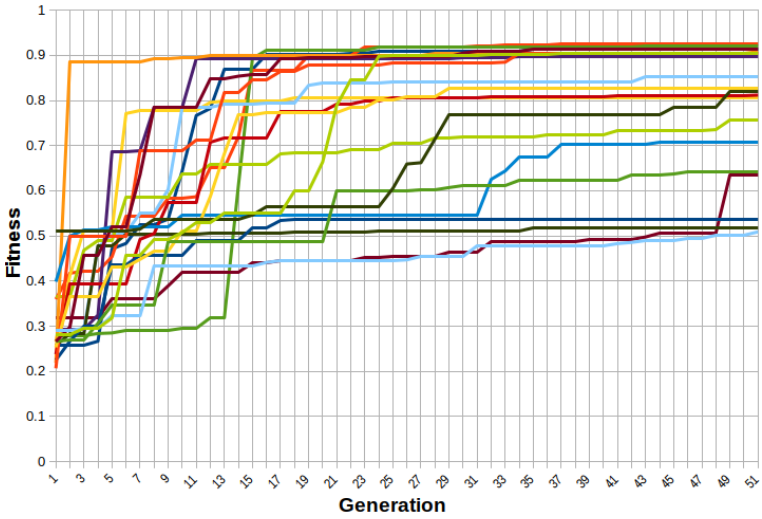


Figure 4.1: Graph showing the fitness of the best individual in each generation from twenty runs.

4.2 Effects of Scaling on Fitness

I have used the best individual with fitness of 0.924 to generate networks with higher density substrates. The idea is that the evolved CPPF encoding represents the general connectivity pattern and even after scaling the network should still retain at least some ability to perform the task it was trained on. The individual is described by the following equations:

$$f_i(r_1, \varphi_1, r_2, \varphi_2) = [(r_2 \cdot \varphi_2) + tg^{-1}(\varphi_2)] \cdot (\varphi_1 \cdot \sqrt{|r_2|}) + cos(tg^{-1}(-1) \cdot \varphi_1^2) \quad (4.1)$$

$$f_n(r_1, \varphi_1, r_2, \varphi_2) = e^{-(e^{-[tg^{-1}(e^{-(\varphi_1 - \varphi_2)^2})]^2 - r_1)^2}} \quad (4.2)$$

$$f_b(r, \varphi) = 0.89537 \quad (4.3)$$

The threshold for connection weights was set to 0.1 during all the evolutions (the effect this has on network generation is described in 3.2.4). When the neural substrate was scaled up the performance of the network dropped slightly, but the robot was still able to follow the road. To the contrary, when the sensor substrate was scaled up, the robot was no longer able to stay on the road. It only drove straight. I then tried to change the threshold to different values and repeat the scaling. For lower values the performance still dropped but the robot was able to navigate successfully and avoid the grass. Interestingly, for some values the performance actually had increased. In Table 4.2 I present the fitness of the evolved individual when various substrate resolutions and threshold values are used. It is noticeable that as the number of sensors increases the trajectory gets smoother and the robot tends to stay in the middle of the road (see Figure A.3).

Substrate		Weight Thresholds		
Sensor	Neural	0.1	0.02	0
3x5	3x3	**0.924**	**0.880**	**0.912**
3x5	25x50	**0.851**	**0.876**	**0.912**
25x50	3x3	**0.116**	**0.914**	**0.944**
25x50	25x50	**0.116**	**0.772**	**0.941**
3x5	3x50	**0.851**	**0.881**	**0.881**
3x5	25x3	**0.884**	**0.908**	**0.910**
200x500	3x3	**0.116**	**0.116**	**0.944**
500x1000	3x3	**0.118**	**0.118**	**0.944**

Table 4.2: Table shows the fitness of the network generated with various substrates and connection weight thresholds.

4.3 Evolution with Various Sensor Substrate Resolutions

The experiment in the previous section has showed that it is possible to scale up the substrates while retaining the performance. This experiment compares the convergence of the evolution when it is run using sensor substrates of various resolution from the beginning. Four different sensor substrates were used while the neural substrate remained the same (see Table 4.3).

Substrate	A	B	C	D
Sensor	3x5	6x13	10x31	20x50
Neural	3x3	3x3	3x3	3x3

Table 4.3: Parameters used in experiment 4.3.

The Figure 4.2 shows the averaged fitness of the best individuals for each of the four sensor substrate resolutions. For the version **A** the data from the first experiment were used, because the setup is the same. The other setups **B**, **C** and **D** have been run 10 times each. It can be seen that **A**, **B** and **C** have comparable convergence while the convergence of **D** is noticeably worse. After 50 iterations average fitness reached was 0.785, 0.732, 0.785 and 0.576 for **A**, **B**, **C** and **D** respectively. The number of iterations (median) before target fitness 0.7 was reached is 14.5 for **A**, 25.5 for **B** (6 runs reached the target fit.), 38 for **C** (7 runs reached the target fit.) and in **D** only three runs reached the target fitness.

Anothers important fact is the time it takes to run the evolution.The time grows approximately linearly with the number of sensors. The average duration of a single run to finish 50 generations was approximately 15 minutes for **A**, 35 minutes for **B**, 2h for **C** and 6.5 hours for **D** respectively.

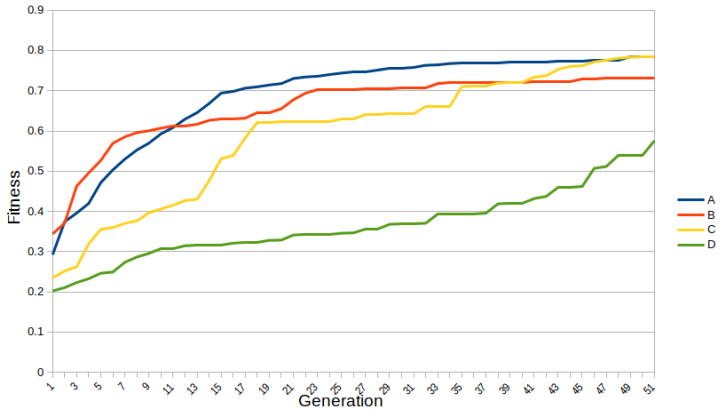


Figure 4.2: Graph showing the convergence of the evolution using different sensor substrates. It shows averaged best fitness from all the runs for each setup.

4.4 Evolution with Gradual Substrate Scaling

The last experiment compares performance of the evolution using dense substrates from the beginning (setup **E**) with the evolution where the substrate density is increased in steps (setup **F**). I have implemented the algorithm in such a way that it is possible to specify in the configuration file the different substrates to use and the rules for when to change the substrates during the

evolution. In practice they are in the form **if** *certain fitness is reached* **then** *use next substrate in the list.*

In the setup **E** both the sensor and the neural substrates are set to 10x31. The different substrates from setup **F** and the transition fitness thresholds are shown in Table 4.4. For this experiment individual with the fitness of 0.9 was considered to be a success.

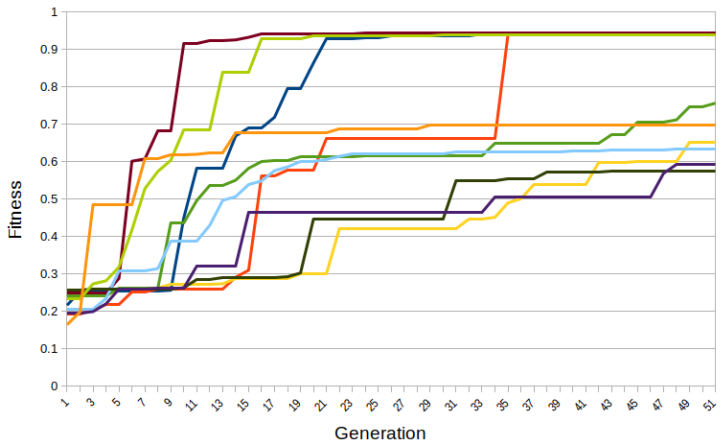
fitness threshold	initial	0.6	0.75	0.8	0.9
sensor substrate	3x5	4x10	6x15	10x31	-
neural substrate	3x3	4x10	6x15	10x31	-

Table 4.4: Parameters used in the setup **F**. The last column is a termination condition which stops the evolution when the fitness of 0.9 is reached using the substrate with the highest resolution.

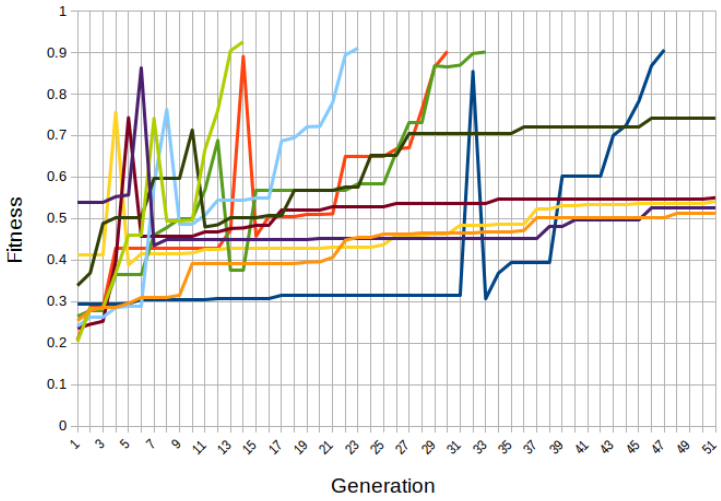
Both of the setups were run ten times each. During the evolutionary runs using setup **E** only four out of ten achieved fitness higher then 0.9. The mean fitness after 50 generations was 0.768. The ten runs are graphed in Figure 4.3 **a)**.

For the second part of this experiment I decided to run the algorithm with connection weight threshold set to 0.01 as the lowering of this parameter yielded better scaling performance in 4.2. From the ten runs five have finished with the highest resolution substrates (10x31) and reached fitness over 0.9 while four have reached with the second substrate resolution (4x10) and one remained with the simplest substrate. Every time the substrate was scaled up the fitness decreased and did not always recover. (see Figure 4.3 **b)**)

If we consider only the runs that reached fitness 0.9, the runs using **E** did so on average in 20.5 hours and runs using **F** on average in 20.7 hours. The difference is not significant. Although runs using **F** started off with low density substrates (which take less time to simulate) they reached the fitness of 0.9 later then runs using **E**.



(a) : Setup **E**.



(b) : Setup **F**.

Figure 4.3: Graph shows convergence of the ten evolutionary runs using the two setups. **a)** shows the evolution with the highest density substrate only. **b)** shows the evolution with gradual increase in substrate density.

Chapter 5

Conclusion

The main goal was to compare different approaches to evolution of neural networks to control a simulated robot. HyperGP algorithm was implemented and the network was encoded as three CPPFs. This encoding has the advantage of being able to generate networks at different scales using the same genome.

The scaled up networks were still capable of driving the robot successfully. The fitness of the network actually increased in some cases when the value of the connection weight threshold (used in network generation process) was lowered. The original network produced by the evolution had only 15 sensors and 9 neurons with fitness of 0.924. Then networks were generated using different substrates. One network consisting of 1250 sensors and 1250 neurons was generated from the genome and its fitness was actually increased to 0.941. The evolution was also run with sensor substrates of different scales. On average the evolution with lower density substrates converged faster.

The last experiment was set up to compare two approaches to the evolution. One where dense sensor and neural substrates were used from the beginning and second where the substrates were progressively scaled up during the evolution. The gradual scaling was set up in a way where a certain fitness had to be reached in order for the substrates to be scaled. Using this gradual scaling approach the fitness of 0.9 was on average reached in later generations then the first approach. But because it takes more time to run the simulation at higher density substrates the two approaches reached the target fitness in the same amount of time and no significant advantage has been found in one approach over the other.

In future when these experiments are performed with different parameters the results might yield better results. For example for the gradual scaling approach only one set of rules was used here and different sets should be tried to find better combinations.

References

[1] GAUCI, Jason and Kenneth O. STANLEY. Generating Large-Scale Neural Networks Through Discovering Geometric Regularities. In: *GECCO 2007: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. New York: ACM Press, 2007, pp. 997–1004.

[2] D'AMBROSIO, David B. and Kenneth O. STANLEY. A Novel Generative Encoding for Exploiting Neural Network Sensor and Output Geometry. In: *GECCO 2007: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. New York: ACM Press, 2007, pp. 974-981.

[3] BUK, Zdeněk, Jan KOUTNÍK and Miroslav ŠNOREK. NEAT in HyperNEAT Substituted with Genetic Programming. In: KOLEHMAINEN, Mikko et al., eds. *Adaptive and Natural Computing Algorithms: 9th International Conference, ICANNGA 2009*. Heidelberg: Springer Verlag Berlin, 2009, pp. 243-252.

[4] HAYKIN, Simon. *Neural Networks - A Comprehensive Foundation*. Prentice Hall, 1998. 2nd ed.

[5] MIIKKULAINEN, Risto and Kenneth O. STANLEY. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*. 2002, vol. 10, no. 2, pp. 99–127.

[6] GOMEZ, Faustino, Jan KOUTNÍK and Jürgen SCHMIDHUBER. Compressed Network Complexity Search. In: COELLO COELLO, Carlos et al., eds. *Parallel Problem Solving from Nature - PPSN XII: Proceedings, Part I*. Heidelberg: Springer Verlag Berlin, 2012, pp. 316-326.

[7] GOMEZ, Faustino, Jan KOUTNÍK and Jürgen SCHMIDHUBER. Evolving Neural Networks in Compressed Weight Space. In: *GECCO 2010: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. New York: ACM Press, 2010, pp. 619-626.

[8] DRCHAL, Jan, Jan KOUTNÍK and Miroslav ŠNOREK. HyperNEAT Controlled Robots Learn How to Drive on Roads in Simulated Environment. In: *Proceedings of Congress on Evolutionary Computation, CEC 2009*. Piscataway: IEEE Press, 2009, pp. 1087-1092.

[9] WILLIAMS, Ronald J. and David ZIPSER. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*. 1989, vol. 1, no. 2, pp. 270-280.

[10] BUK, Zdeněk. *Continual Evolution Algorithm*. Prague, 2012. Dissertation thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science and Engineering.

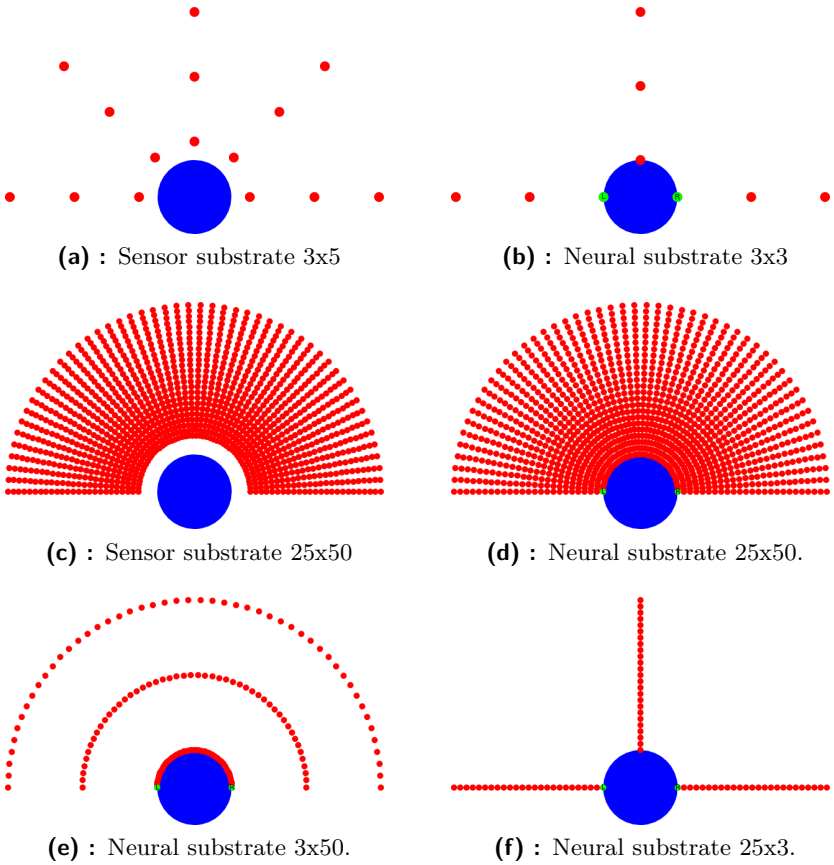
[11] LANGDON, William B., Nicholas F. McPHEE and Riccardo POLI. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008.

[12] GOLDBERG, David E. and Brad L. MILLER. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems*. 1995, vol. 9, pp. 193-212.

[13] KOZA, John R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Stanford University Computer Science Departement Technical Report, 1990.

Appendix A

Additional Figures



(a) : Sensor substrate 3x5

(b) : Neural substrate 3x3

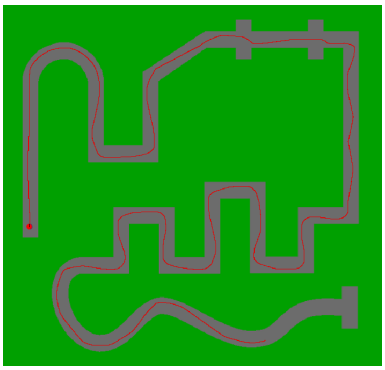
(c) : Sensor substrate 25x50

(d) : Neural substrate 25x50.

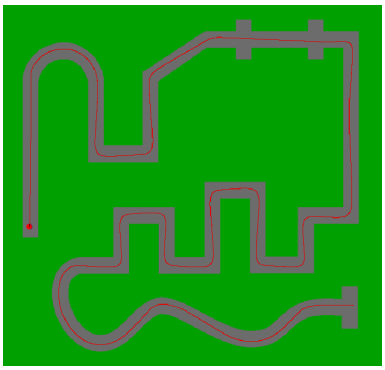
(e) : Neural substrate 3x50.

(f) : Neural substrate 25x3.

Figure A.1: Substrates at various resolutions.

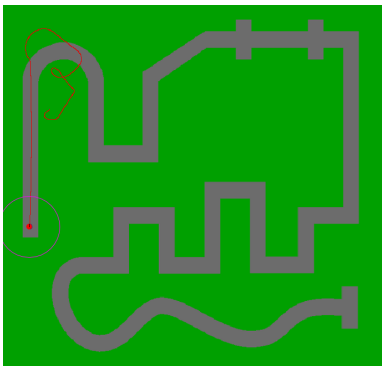


(a) : Sensor substrate 3x5, neural substrate 3x3.

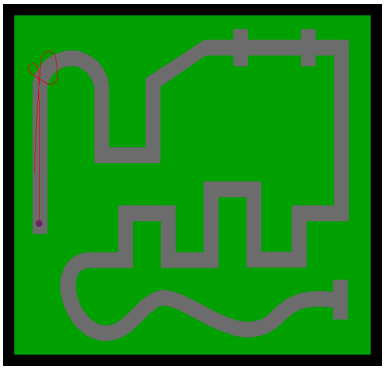


(b) : Sensor substrate 25x50, neural substrate 3x3

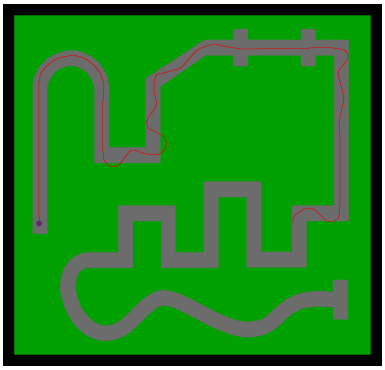
Figure A.2: Trajectories produced by the network generated by same genome but at two different sensor substrate resolutions.



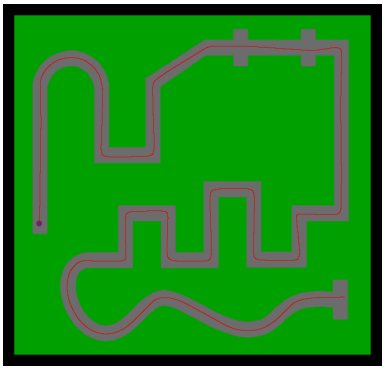
(a) : Trajectory of an individual from an initial population.



(b) : This individual tries to stay on the road but can turn only to one side.



(c) : Here the individual manages to follow the road but sometimes fails.



(d) : This is one of the best evolved individual's ability to drive is almost perfect.

Figure A.3: Trajectories traveled by networks at different stages in evolution.

Appendix B

Manual

B.1 Overview

Figure B.1: Starting menu.

Once the program is started we have four options (Figure B.1) a) create a new project, b) create a new project and load configuration from a file, c) open sandbox where the user can experiment on an evolved individual providing it with different substrates and d) open a tool to visualize the three CPPFs. When choosing the options a) or b), we are taken to the main window (Figure B.2). The only difference is that choosing a) fills the parameter settings with default values while choosing b) will load files from a configuration file.

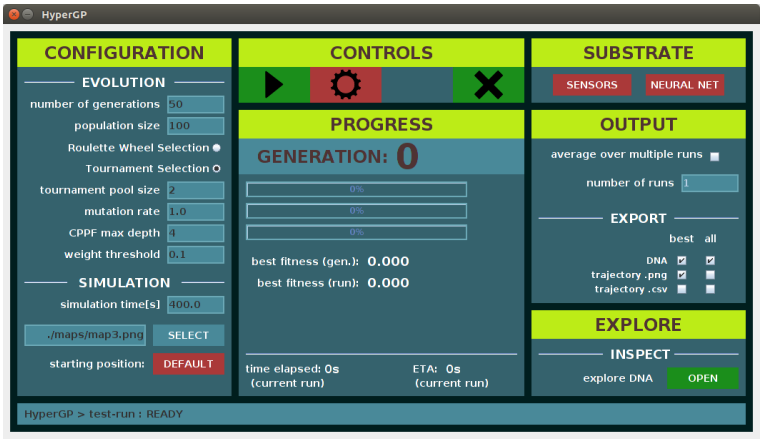


Figure B.2: Main window.

On the left side is the configuration panel containing most of the parameter options. On the right we can specify the substrates, select export options or visualize the three CPPFs. The panel in the middle shows the progress during the evolution.

When we are running evolution using single substrate, we can change it manually. First pause the evolution and after the generation has been finished, click on the configure button (gear icon). This will enable the alteration of the substrates. Choose one substrate in the top-right and the substrate editor will appear.

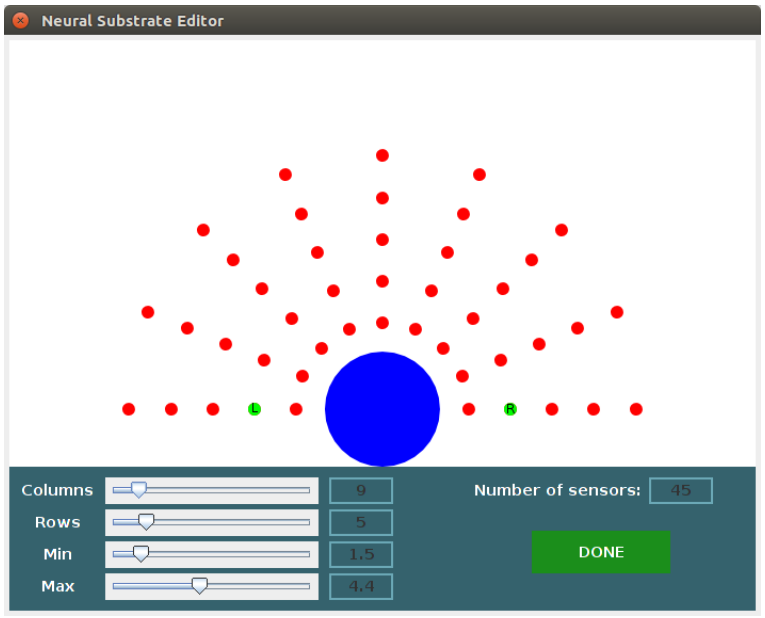


Figure B.3: Substrate editor.

The substrate editor lets us specify the number of sensors or neurons and their layout around the robot. It is possible to change the number of units in the substrate and change the area the substrate covers. In the gui the parameters have a maximum value. This can be circumvented by using a configuration file. There we can specify arbitrarily large substrates. When we work with neural substrate, we need to select the output neuron. This is done by using left or right mouse button to click on one of the neurons.

In map placer panel we can drag the robot (red circle) across the map, rotate it by dragging the circle shown around it and scale it by scrolling.

In sandbox we can take evolved genome measure its fitness on different maps using various substrates.

B.2 Configuration File

The configuration file is simply a text file with filename extension **.cfg**. All the parameters have a human readable names. There are two sample files in the /hypergp directory on the CD. The *sample.cfg* describes a normal

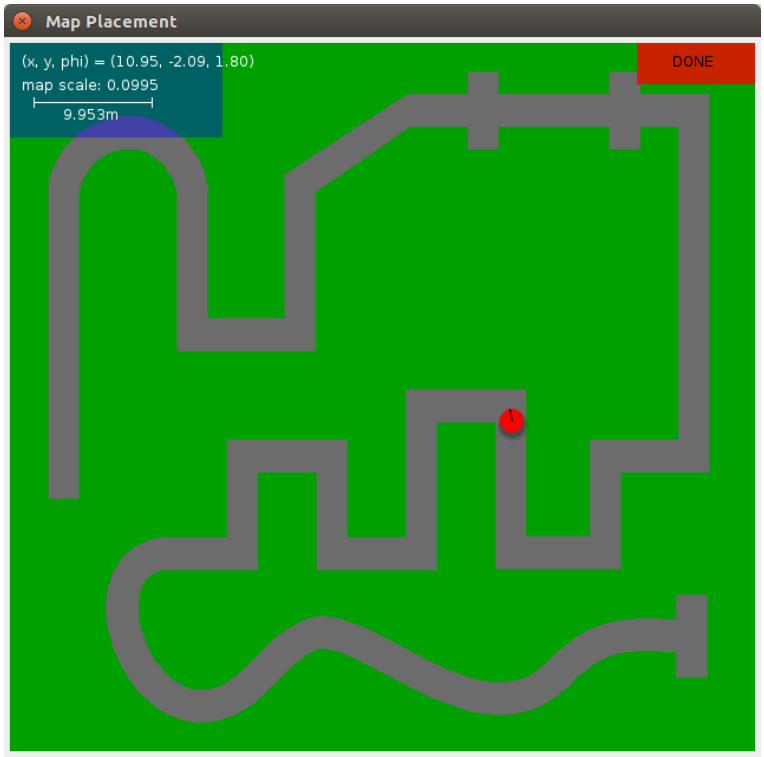


Figure B.4: Map placer.

run while *sample-scaling.cfg* describes run using the rules for scaling the substrates up. There are configuration files for each experiment.

B.3 DNA File

The files with the filename extension **.dna** are used to store the evolved genomes. The DNA file is human-readable. It contains three CPPFs in a textual form. A sample is shown in Figure B.6. first line defines maximum depth of the CPPF expression and the secon te number of variables the CPPF can use. The next line is the root of the CPPF tree. As we progress line by line we in fact traverse the CPPF tree in depth-first fashion.

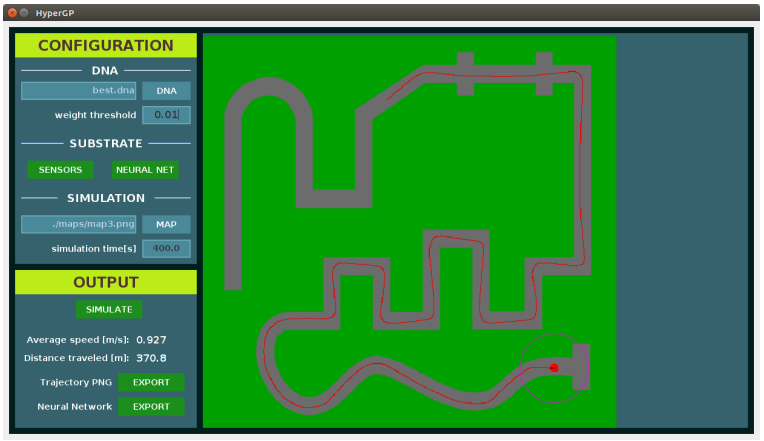


Figure B.5: Sandbox window.

```
4
4
v2
###
4
4
.GAUSS1
.SQUARE
.PLUS
.TIMES
v2
v2
.SIN
v1
###
4
2
.ATAN
.PLUS
.TIMES
.GAUSS2
v0
c1.6846366045952177
.TIMES
v1
c-1.0
.SQUARE
.COS
c-1.0
```

Figure B.6: A file encoding three CPPFs.

Appendix C

Code Samples

C.1 Neural Network Evaluation Method

```
/**
 * Method to calculate the outputs of the network given the input
 * values and its state.
 * @param inputs - array of input signals
 * @return an array containing activations of output neurons
 */
public double[] evaluate(double[] inputs) {

    double[] out = new double[numOfOutputs];

    for(int i=0; i<numOfNeurons; i++){
        double sum = biases[i];
        for (int j = 0; j < numOfInputs; j++) {
            sum += inputs[j] * inputWeights[i][j];
        }
        for (int j = 0; j < numOfNeurons; j++) {
            sum += state[j] * recurrentWeights[i][j];
        }
        newState[i] = sigmoidUnipolar(sum);
    }
    arraycopy(newState,0,state,0,numOfNeurons);

    for(int i=0; i<numOfOutputs; i++){
        out[i] = state[outputs[i]];
    }

    return out;
}
```

Appendix D
Hardware and Software

D.1 Hardware

CPU: Intel(R) Core(TM) i5 M450, 2.4GHz 4 cores, 8 threads
Memory: 3695MiB
Graphics Card: GT218M [GeForce 310M]

D.2 Software

Operating System: Ubuntu 17.04 32-bit, Linux 4.10.0-21-generic
Java 1.8.0
Netbeans IDE 8.2
TeX Live
TeXMAKER 4.5

Appendix E

CD contents

- **/dist** - directory contains executable JAR file and javadoc
- **/experiments** - the files generated during the experiments along with configuration files
- **/hypergp** - netbeans project directory containing all the source files
- **/thesis** - contains this document as PDF and LaTeX source files.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: David P a v l í č e k

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Scalable Representations of Neural Networks

Guidelines:

1. Review methods for neuroevolution using indirect encoding of neural networks with focus on network and sensor resolution scalability.
2. Create a simulation environment to carry out the experiments with two-wheeled robots controlled by neural networks.
3. After discussion with the thesis supervisor choose an appropriate method and experiment on two-wheeled robots.
4. Replicate the two-wheeled robot experiments from [1].
5. Extend the experiment by scalable input sensors and test the ability of the network to train with variable resolution of the sensors.
6. Compare the evolution of full-resolution network with evolution which uses gradually increasing number of sensors and neurons. Focus on evolution time and final solution quality.

Bibliography/Sources:

[1] Z. Buk, J. Koutník, M. Šnorek: Neat in hyperneat substituted with genetic programming. ICANNGA 2009.
[2] F. Gomez, J. Koutnik, J. Schmidhuber: Compressed Network Complexity Search, PPSN 2012.
[3] D. B. D´Ambrosio and K. O. Stanley: A novel generative encoding for exploiting neural network sensor and output geometry. GECCO 2007.

Bachelor Project Supervisor: Ing. Zdeněk Buk, Ph.D.

Valid until: the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 17, 2017